Software Engineering Institute

Using Aspect-Oriented Programming to Enforce Architecture

Paulo Merson

September 2007

TECHNICAL NOTE
CMU/SEI-2007-TN-019

Software Architecture Technology Initiative
Unlimited distribution subject to the copyright.

Carnegie Mellon

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2007 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

Abstract

Using aspect-oriented programming (AOP), software developers can define customized compile-time error or warning messages that are issued when the code contains join points that match specified pointcuts. These customized messages are generated by compile-time declarations, which are an extremely simple but powerful AOP mechanism. Declarations that look for nonvalid interactions between modules can be used for architecture enforcement. Coding policies, best practices, design patterns, and code-naming conventions can also be enforced. Compile-time declarations operate as an additional verification in the build process, but they do not affect the compiled application and can be turned on and off at any time. That feature makes this approach an automated and nondisruptive solution for architecture enforcement and a risk-free first step towards AOP adoption.

1 Introduction

Aspect-oriented programming (AOP) is a programming paradigm that facilitates modularization of crosscutting concerns. The AOP term and concept originated at Xerox PARC in the 1990s [Kiczales 1997]. AOP is gathering momentum in the software engineering community. On the research front, researchers actively investigate issues in the broader discipline of aspect-oriented software development. Research topics include type systems for aspects, composition models and operators for aspects, architecture design, requirements engineering, and the modeling and visualization of aspects. On the practitioner front, tools, frameworks, and aspect libraries are evolving fast with respect to usability and reliability. An active community of developers is enjoying the benefits of AOP in projects that span various business segments and development platforms.[1] Practitioners discover new uses for aspects every day.

The goal of this report is to show, through examples, how you can use AOP to ensure

- conformance to architectural design
- the proper use of design patterns and programming best practices
- conformance to coding policies and naming conventions

The audience for this report consists of architects and developers who are familiar with AOP concepts. All the examples use the AspectJ syntax[2] [Xerox 2003].

The report is structured as follows: Section 2 describes the static AOP compile-time declaration mechanism. Section 3 briefly introduces the architecture conformance challenge and then shows how compile-time declarations can be used to enforce architectural constraints. Section 4 provides various examples of coding policies and best practices that can be enforced with AOP. In addition, that section describes how AOP can enforce naming conventions. Section 5 provides some concluding remarks.

[1] You can find examples of applications of AOP in the industry track of the annual Aspect-Oriented Software Development (AOSD) Conference and in emails to the aspectj-users@eclipse.org mailing list. To access those emails, go to http://www.eclipse.org/aspectj/userlists.php.

[2] To implement and test the examples shown in this report in your Java project, follow these steps:

- Install AspectJ on your machine.
- Copy and paste all code snippets into a single public aspect (e.g., `public aspect Enforcement {…}`). Then, save the file—for example, as Enforcement.aj.
- Change the aspect code to target the packages of your project where applicable. (The examples in this report use `com.foo.proj`.)
- Compile the Java code and the aspect together using the AspectJ compiler.

2 Compile-Time Declarations

AOP mechanisms can use dynamic or static crosscutting. With dynamic crosscutting, at compile time or load time, aspect code is added to the target units through weaving at specified join points. Logging is a typical example of a crosscutting concern that can be implemented using dynamic crosscutting—calls to log methods are inserted through weaving at the beginning of methods whose execution should be logged. Dynamic crosscutting adds or modifies the executable code and hence the behavior of a program. In this report, we won't use dynamic crosscutting.

Static crosscutting modifies the static structure of the types in the application and their compile-time behavior [Laddad 2003]. It can be used, for example, to

- add a method `void init(ServletConfig config)` with standard initialization code to all classes that implement the `javax.servlet.Servlet` interface in a given project. This mechanism is usually referred to as intertype member declaration [AspectJ 2003, Gradecki 2003] or member introduction [Laddad 2003].
- make all classes whose name ends in the letters "PK" (for "primary key") implement the `java.io.Serializable` interface. This static crosscutting mechanism is called type-hierarchy modification [Laddad 2003].
- treat the checked exception `java.io.IOException` as an unchecked exception on all calls to `java.io.FileInputStream.close()`. This mechanism is exception softening [Gradecki 2003, Laddad 2003].

The other application of static crosscutting is the introduction of compile-time errors or warnings when join points that match the specified pointcut are found. This mechanism is generally called compile-time declaration or custom compilation messages and is the AOP mechanism used in this report for architecture enforcement. As an example, suppose you are using JUnit[3] for automated unit testing and a policy states that all test case classes should have the prefix "Test." The code snippet below using AspectJ syntax causes the compiler to issue a warning if it finds any class under package `com.foo.proj` that does not follow that rule:

```
declare warning :
    staticinitialization(junit.framework.TestCase+) &&
    !staticinitialization(com.foo.proj..Test*) :
    "JUnit test cases should start with 'Test'";
```

Declaring compile-time errors and warnings this way is less intrusive, because the target code is not modified in any form. No new code is woven as in dynamic crosscutting, and no type is altered as in intertype member declaration or hierarchy modification. This fact brings a special value to compile-time declarations. If they are added to a project, they can be turned on and off, and the compiled code remains the same.

[3] For more information about JUnit, go to www.junit.org.

3 Enforcing the Architecture

The diagram in Figure 1 shows the top-level decomposition of an application into four layers. The architecture follows the basic design principle of separation of concerns. The User Interface layer has modules that render the screens and handle presentation logic and dialog flow. The implementation of this layer will vary substantially depending on the technology used (e.g., Web-based user interface [UI], Web 2.0, Windows application, Eclipse-based UI). The Core Logic layer contains the modules that implement the business logic of the system and that stay less dependent on the technology. Modules in the Data Access layer implement the logic to access the relational database, including object-relational mapping and classes that contain SQL statements. This layer allows the Core Logic layer to be independent of table schemas and peculiarities of types of databases. Finally, the JDBC layer is the standard Java Database Connectivity (JDBC) application program interface (API).[4] It consists of off-the-shelf libraries that can be used uniformly to access different relational databases, such as Oracle or Microsoft SQL Server.

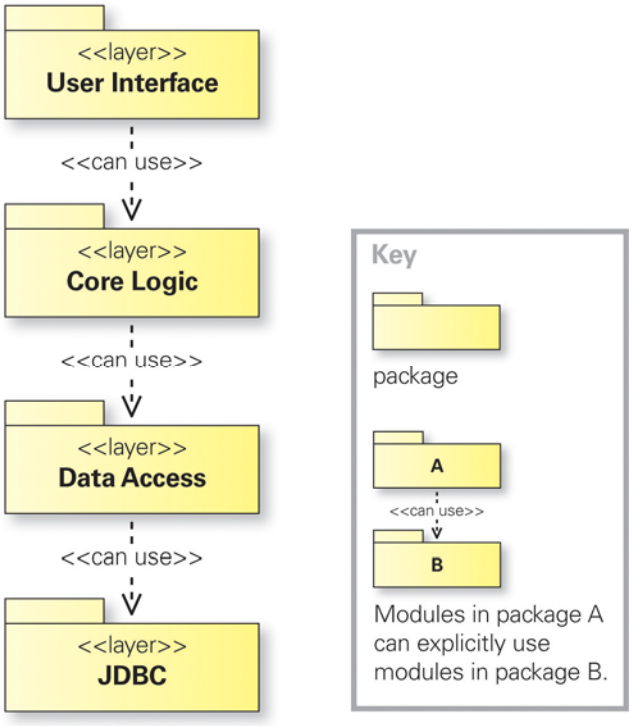

Figure 1: Modules in a Layered Architecture

The dependency between layers is labeled as "can use." This is the typical relation in layered designs and represents the fact that a module in the upper layer is allowed to use any of the public facilities provided by the lower layer [Clements 2003]. The "can use" relation is flexible—it doesn't identify dependencies between specific modules that live inside each layer. In subsequent refinements of the architecture, these dependencies become explicit. Nonetheless, the top-level

[4] For more information, go to http://java.sun.com/jdbc.

architectural design in Figure 1 imposes important restrictions: a module inside the User Interface layer is not allowed to use a module in the Data Access or JDBC layers, a module in Data Access can't use a module in Core Logic, and so on. The layered architecture was created by the architect to satisfy modifiability, portability, and testability requirements. If the code introduces layer bridging that is not conformant to the architecture, these goals may be compromised.

During implementation and maintenance, programmers sometimes introduce dependencies in the code that don't follow the original architectural design. Enforcing that the code continues to conform to the architectural design is a major challenge, and, in fact, failing to do it causes many common software problems [Brown 1998]. There are at least five approaches that help to enforce the architecture or at least check for conformance between architecture and code:

- **code inspections:** Code reviews have a very positive impact on software quality and are more efficient than testing with respect to detecting defects [Humphrey 1995]. However, this is a manual process. Extensive code reviews for checking if the code follows the architecture take time and require the reviewer to have a solid understanding of the architecture, which is not always the case.
- **architecture reconstruction:** This consists of obtaining architectural representations by extracting information from implementation artifacts (e.g., source code, deployment descriptors) or traces of the system execution [Kazman 2002]. Reconstructed architectural views can then be compared with the original intended design to identify mismatches. Recovering the architecture to verify conformance with the original design is costly, but architecture reconstruction has other benefits, such as producing detailed and up-to-date architecture depictions.
- **model driven architecture (MDA):** If the MDA process (as described by Kleppe, Warmer, and Bast [Kleppe 2003]) is followed, code is generated by an MDA tool based on designs typically expressed in UML. Even if the code is later modified directly, the tool usually allows reversing it back to design without losing the modifications. Therefore, in theory, architecture conformance is easy to achieve, because code and design can be kept in synch by the MDA tool.
- **enforcement tools:** Tools that help enforce that the implementation follows the architecture design are already available. Examples include Lattix,[5] Sotograph,[6] and Structure101.[7]
- The other alternative, which will be described next, is the use of AOP.

3.1 ENFORCING ARCHITECTURAL CONSTRAINTS USING AOP

AOP lets us specify locations in the source code called *join points*. Some examples of join points are the invocation of a method or constructor; the declaration of a class, method, or constructor; and access to a member variable of a class. Wildcard patterns can be used to express a set of join points in the target code. For example, `call(* com.foo.proj..*.set*(String))` represents all calls to methods that

[5] For more information on Lattix, go to www.lattix.com.

[6] For more information on Sotograph, go to www.software-tomography.com.

[7] For more information on Structure101, go to www.headwaysoftware.com.

- return any data type
- reside in any class that is part of package `com.foo.proj` or any subpackage
- start with "set" (e.g., `setName`)
- take a `String` object as an argument

There are also constructs that delimit a lexical scope in the code. For example, `within(com.foo.proj.ui..*Dialog)` represents the code in all classes that

- reside in package `com.foo.proj.ui` or any subpackage
- end with `Dialog` (e.g., `PlaceOrderDialog`)

These AOP mechanisms can be used to check whether there are relations in the code not prescribed by the architectural design. Going back to the example in Figure 1, the layers will eventually be implemented in Java as a set of Java packages. Figure 2 shows the same layered design with the actual names of the Java packages implementing the layers.

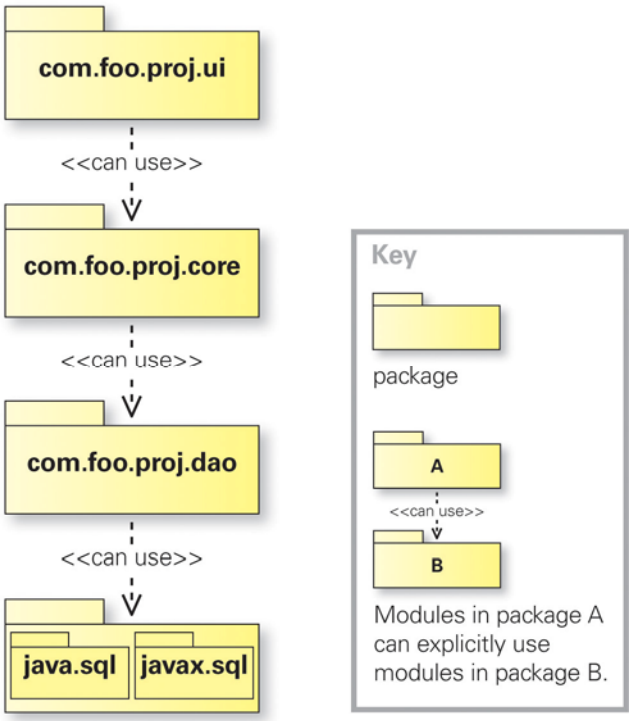

Figure 2: Layered Design from Figure 1 Showing the Corresponding Java Packages

Knowing the design restrictions imposed by the original layered design in Figure 1 and knowing how the layers map to Java packages in the code base (Figure 2), it is possible to create compile-time declarations to enforce the layered design. For example, the following aspect checks at compile time that business logic modules in the Core Logic layer do not make explicit calls to UI modules:

```
public aspect Enforcement {
    public pointcut inCore() : within(com.foo.proj.core..*);
```

```
    public pointcut callToUi() : call(* com.foo.proj.ui..*+.*(..)) ||
                                 call(com.foo.proj.ui..new(..));

    declare warning : inCore() && callToUi() :
        "Core logic layer can't have calls to UI layer";
}
```

In this aspect, there are two *pointcuts*: `inCore` and `callToUi`. A pointcut is simply a named construct that describes a set of join points. Pointcuts can be referred to in compile-time declarations and other AOP constructs. The first pointcut (`inCore`) defines a scope in the code base that consists of all the code inside package `com.foo.proj.core` or any subpackage. Pointcut `callToUi` has two parts. The first part refers to calls to any methods in the `com.foo.proj.ui` package or subpackages. The second part refers to calls to any constructors (keyword `new`) in the same set of packages. The compile-time declaration is the statement that starts with `declare`. It determines that, if there is a call to a class in the UI layer anywhere in core logic packages, the compiler will show a warning on that call. To be more strict with the enforcement rules, we can use `declare error` instead of `declare warning` and generate a compile error.

Similar pointcut definitions and declare statements can be added to verify that only the dependencies depicted in Figure 2 are present in the code. Then, every time the application is built, the compiler will issue warnings if there are disallowed calls.

In addition to the architectural design, component technologies have constraints that must be satisfied by the components. These contractual obligations ensure that independently developed components can interact in predictable ways and can be deployed into standard runtime environments [Bachmann 2000]. Take, for example, the Enterprise JavaBeans (EJB) component technology. The specifications [Sun 2001] determine that a stateless session bean class must define a single `ejbCreate()` method that takes no arguments. Such a rule is usually enforced by a deployment tool that is part of the Java 2 Platform, Enterprise Edition (J2EE) application server suite. Other rules and restrictions are usually stated in the specifications but are not enforced by the compiler or deployment tool. For example, an EJB must not make graphical user interface (GUI) calls, must not read or write to files in the file system, must not manage threads, and must not make calls to native code. Most of these restrictions can be checked using AOP [Laddad 2003]. The following declaration can help to prevent the use of native code in EJB classes:

```
public pointcut inEJB() : within(javax.ejb.EnterpriseBean+);

public pointcut callNative() :
    call(* System.loadLibrary(..)) || call(* System.load(..)) ||
    call(* Runtime.loadLibrary(..)) || call(* Runtime.load(..)) ||
    call(native * *.*(..));

declare error : inEJB() && callNative() :
    "EJBs cannot load native code";
```

3.2 A CONCRETE EXAMPLE

The J2EE 1.3 Tutorial published by Sun Microsystems [Bodoff 2007] includes an example of a multitier application called Duke's Bank. Figure 3, a graphical representation of the Runtime view

of that application's architecture, was adapted from that tutorial. At runtime, the Web client and the application client call the session beans, the session beans invoke the entity beans, and the entity beans access the database tables on the back end. Restricting all database access to entity beans has some benefits. Portability and modifiability are improved, because changes related to porting to a new database or altering the structure of the database tables are confined to the entity beans.

Assuming that constraint was the intent of the architect, we can create a compile-time declaration to check that all database calls occur within the entity beans:[8]

```
public pointcut inEntityBean() : within(javax.ejb.EntityBean+);

public pointcut callToJdbc() : call(* java.sql..*+.*(..)) ||
                               call(java.sql..new(..)) ||
                               call(* javax.sql..*+.*(..)) ||
                               call(javax.sql..new(..));

declare warning : !inEntityBean() && callToJdbc() :
    "Only entity beans should access the database";
```

The `inEntityBean` pointcut delimits the scope of all entity beans. The wildcard pattern `EntityBean+` refers to any class that implements the `EntityBean` interface.[9] This way, we get all entity beans in the code that will be compiled. Database calls would use the JDBC API and are caught by pointcut `callToJdbc`. The compile-time declaration gives a warning if there is a JDBC call that is not inside an entity bean.

Surprisingly, the compile-time declaration above applied to the tutorial source code reveals a discrepancy between the code and the design in Figure 3. In the implementation, the session beans also access the database directly. Perhaps, these "undesigned" calls were created, because the developer opted to avoid entity beans by using the JDBC for Reading pattern [Marinescu 2002] to improve performance for some operations. In any case, the declaration reveals an inconsistency between the architectural design and the code.

[8] In this example and those that follow, the surrounding `public aspect` declaration is removed to save space.

[9] Character '+' following an identifier may also denote "any subclasses" if that identifier is a class name.

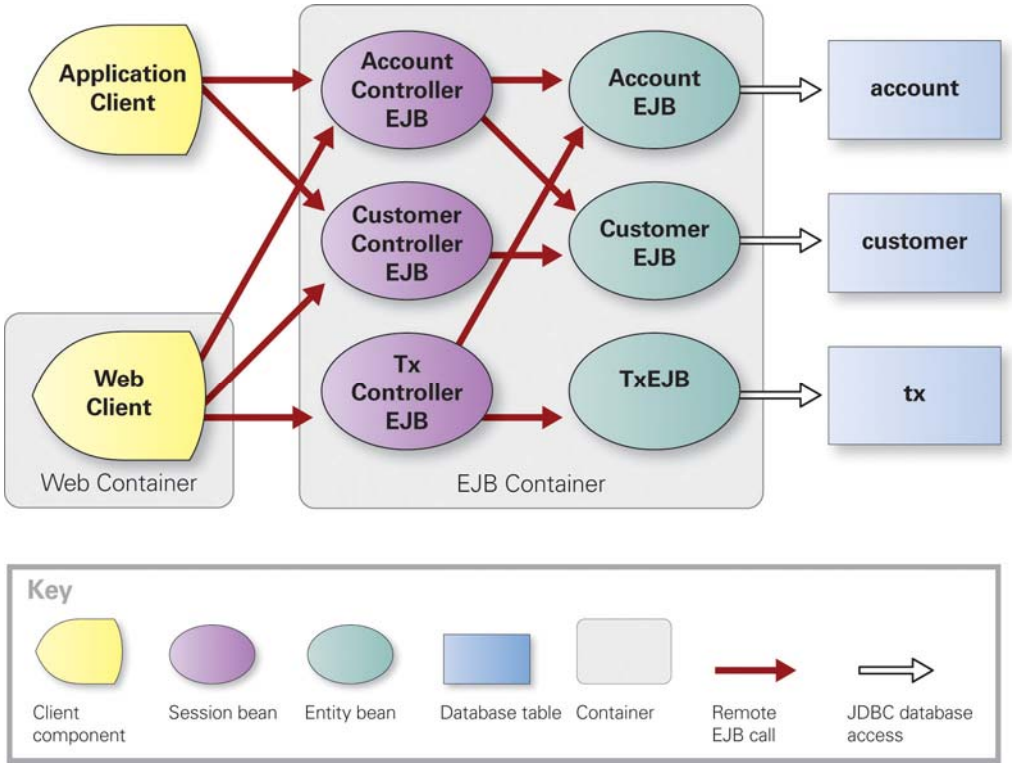

Figure 3: Runtime View of the Architecture of Duke's Bank Application [Bodoff 2007]

3.3 ENFORCING PATTERNS

Some design patterns can also be enforced using AOP compile-time declarations. Figure 4 shows a UML class diagram that exemplifies the application of the Abstract Factory design pattern [Gamma 1995]. Class `SomeScreen` represents a screen of an application that should be portable across the Java Swing and SWT[10] user interface frameworks. `SomeScreen` uses the `WidgetFactory` abstract class to create instances of the widgets (window, scrollbars, buttons, etc.) that will be displayed to the user. The factory creates the concrete widgets using either the Swing or SWT framework, based on a selection made at initialization or build time. `SomeScreen` and similar classes that instantiate widgets should use the abstract factory, which is what we would like to enforce. If these client classes directly instantiate concrete widget classes or call concrete factories, portability will be impaired. The code snippet below enforces the pattern:

```
public pointcut inFactory() :
    within(com.foo.proj.ui.*WidgetFactory);

public pointcut callBypassingFactory() :
    call(com.foo.proj.ui.Window+.new(..)) ||
    call(com.foo.proj.ui.ScrollBar+.new(..));

public pointcut callToConcreteFactory() :
```

[10] For more information on SWT, go to www.eclipse.org/swt.

```
    call(!abstract * com.foo.proj.ui.WidgetFactory+.*(..));

declare warning :
    callBypassingFactory() && !inFactory() :
    "Use factory to instantiate this class.";

declare warning :
    callToConcreteFactory() :
    "Use abstract factory instead of concrete factory.";
```

Similarly, other patterns that restrict the interactions allowed between elements can be enforced using compile-time declarations. Examples include Mediator [Gamma 1995], Session Façade [Marinescu 2002, Alur 2003], and Data Access Object [Alur 2003].

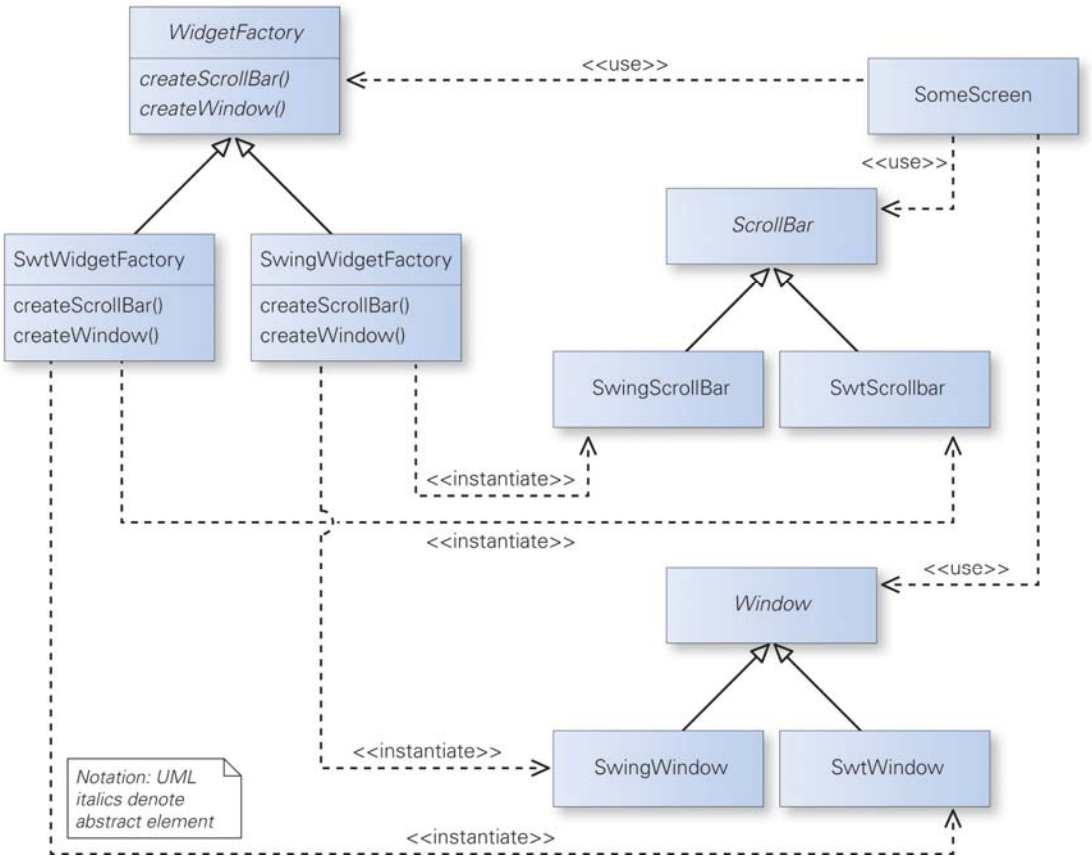

Figure 4: Abstract Factory Design Pattern [Gamma 1995] (Adapted)

4 Conformance to Coding Policies

Numerous implementation policies and best practices can be enforced using compile-time declarations. For example, it is a common convention in Java to add the suffix “Exception” to all subclasses of `Exception`. Here’s how it can be checked using AOP for all subclasses of `Exception` under package `com.foo.proj`:

```
public pointcut misnamedException() :
    execution(Exception+.new(..)) &&
    execution(com.foo.proj..new(..)) &&
    !execution(com.foo.proj..*Exception.new(..));

declare warning : misnamedException():
    "Subclasses of Exception should terminate in 'Exception'";
```

The difference between `execution` and `call` is subtle. The keyword `execution` represents join points at the body of the specified constructor or method. The keyword `call` represents join points wherever the specified method is called. The compile-time declaration above uses `execution` so that it issues a warning on any constructor of the Exception subclass with an illegal name. If it used `call`, the warning would appear on the calls to the constructor and hence would not be seen if the class was not being used yet.

Still with respect to exceptions, in some projects, it is recommended that all exceptions be created with an error message or a `Throwable` object as an argument. The following declaration alerts if any type of exception is created without arguments:

```
public pointcut noArgsException() : call(Exception+.new());

declare warning : noArgsException() :
    "Shouldn’t create exception without cause or message.";
```

It is likely that, in a GUI application, exception stack traces are directed to a log file or handled in some way by the UI layer. Thus, we want to avoid calls to `printStackTrace()` in the code. Likewise, print statements using the default output streams (`System.out`, `System.err`) are not desirable in production code. In practice, we may permit such calls inside `main()` methods that are created in some classes just for tests. Here is the compile-time declaration to check for violations of these conventions:

```
public pointcut callToPrint() :
    call(* java..Throwable.printStackTrace(..)) ||
    call(* System.out.print*(..)) ||
    call(* System.err.print*(..)) ||

public pointcut inMainMethod() :
    withincode(public static void main(String[]));

declare warning : callToPrint() && !inMainMethod() :
    "Print statements should not be in production code.";
```

Another common policy is to access member variables only through get and set methods to improve information hiding. A simple declaration identifies this kind of violation [Laddad 2003]:

```
public pointcut accessPublicVars() :
    get(public !final * *) || set(public !final * *);

declare warning : accessPublicVars() :
    "Consider get/set methods instead of public member variable.";
```

Enforcement declarations can also be used with Java 5.0 metadata annotations. If you are using Apache Beehive[11] to implement Web Services, you add the annotation "@WebMethod" to the methods in your Java class that will be exposed as Web Services. Here's an example:

```
@WebMethod
public double getQuote(@WebParam String symbol) {
    double quote = 0.00;
    // obtain quote...
    return quote;
}
```

The documentation of the @WebMethod annotation indicates that annotated methods must be public. This rule can be enforced using AOP:

```
public pointcut nonPublicWebMethod() :
    execution(@WebMethod !public * *.*(..));

declare error : nonPublicWebMethod() :
    "Methods with @WebMethod annotation must be public";
```

AOP can also help to enforce naming conventions. For example, the usual convention for the name of member variables in Java is to start with a lowercase letter, unless the variable is a constant. The following AOP compile-time declaration ensures that the code does not contain a non-final member variable that starts with a capital letter:

```
public pointcut varStartingWithUpperCase() :
    get(!final * com.foo.proj..*+.A*) ||
    set(!final * com.foo.proj..*+.A*) ||
    get(!final * com.foo.proj..*+.B*) ||
    set(!final * com.foo.proj..*+.B*) ||
    ...
    get(!final * com.foo.proj..*+.Z*) ||
    set(!final * com.foo.proj..*+.Z*);

declare warning :
    varStartingWithUpperCase() :
    "Non-final variables should not begin with capital letter.";
```

[11] For more information on Apache Beehive, go to http://beehive.apache.org/.

The declaration of a member variable is not an exposed join point in AspectJ. For that reason, the pointcut does not target the declaration; instead, it points to any statements where the variable is accessed for read ("get(*signature*)") or write ("set(*signature*)").

Many other policies, rules, or best practices can be enforced with compile-time declarations. Here are some examples:

- If you don't want a specific method or class to be used anymore, but you can't remove it because it is used in legacy code, you can declare an error when it is used outside the scope of the legacy code. The compile-time declaration is more effective than using the "@deprecated" Javadoc tag, which is just a reminder in the code documentation for developers to avoid the tagged element.
- Components that execute in a multithreaded environment (e.g., Servlets) should not store thread-specific state in instance variables. Otherwise, data from one thread can overwrite data from another [Gradecki 2003].
- Sometimes we use a pool of instances to avoid the time-consuming instantiation of objects. Database connections, images, Java Naming and Directory Interface (JNDI) contexts, and EJB home objects are examples of objects that are usually in a pool. Compile-time declarations can enforce that client classes get instances from the pool, instead of creating instances directly.
- Some projects have specific naming rules that can be enforced with compile-time declarations. For example, classes that follow the Data Access Object pattern usually have the suffix "Dao." JUnit test cases usually have suffix or prefix "Test."
- When a concrete class implements an interface, the usual intent is that the outside world will use the contract specified by that interface to interact with objects of that class. However, the class may offer other public operations beyond what is specified by the interface—a common situation when the class implements more than one interface. Compile-time declarations can enforce that a class is only accessed through the interface(s) it implements, so that traceability of contracts doesn't get lost.

5 Conclusion

Over the past 20 years, software engineers became aware that software architecture is critical to success in software projects. Techniques, languages, and patterns were developed to help us create, document, and evaluate architectural designs. Today, architects may have good confidence in the quality of the architectural designs they produce, but there is little confidence that the code created by developers will actually follow the design. When the code deviates from the design, quality attributes such as modifiability and performance can suffer. Architecture enforcement is a major challenge.

When no automated solution is available, some organizations resort to manual code inspections to verify code conformance to the architecture. However, code inspections are prone to human error and don't scale well to large systems and distributed teams. Some commercial tools already promise continuous architecture enforcement. Another approach that solves part of the architecture conformance problem is MDA. Code is generated from UML models and will necessarily follow the design expressed in UML. However, MDA has some barriers to overcome before it becomes mainstream, such as the tendency of software engineers to have syntactic and semantic discipline at the code level and not at the architecture level. At least for Java-based systems, AOP provides a relatively simple automated solution for architecture enforcement. One can create AOP compile-time declarations that will search the entire code base and flag invalid interactions.

In any situation, the first step to be able to enforce the architecture over the lifetime of the system is to have a good architecture representation. If the documentation is incomplete, unclear, or out-of-date, it is hard to apply any architecture conformance technique. More importantly, it is hard for developers to faithfully obey the dictates of the architecture.

The use of AOP for enforcement of coding policies and architecture is a low-hanging fruit that has been explored for a few years and suggested in books, papers, and presentations. Even an open source library with a few examples has been created.[12] This report presented a sample of the variety of applications of compile-time declarations. The code snippets show how compile-time declarations are simple and powerful.

The use of compile-time declaration of errors and warnings is the perfect first step to AOP adoption, because they don't alter the binaries produced during compilation. Therefore, compile-time declarations can be turned on and off at any time, because the original code remains completely independent of the AOP code.

[12] Go to http://patterntesting.sourceforge.net/.

References

URLs are valid as of the publication date of this document.

[Alur 2003]
Alur, D.; Crupi, J.; & Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition.* Upper Saddle River, NJ: Prentice Hall, 2003.

[Bachmann 2000]
Bachmann, F. et al. *Volume II: Technical Concepts of Component Based Software Engineering, Second Edition* (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000.
www.sei.cmu.edu/publications/documents/00.reports/00tr008.html.

[Bodoff 2007]
Bodoff, S. & Jendrock, E. *The Java EE5 Tutorial, Third Edition.* Harlow, England: Addison-Wesley, 2007.

[Brown 1998]
Brown, W. et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* New York: NY: John Wiley & Sons, 1998.

[Clements 2003]
Clements, P. et al. *Documenting Software Architectures: Views and Beyond.* Boston, MA: Addison-Wesley, 2003.

[Gamma 1995]
Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[Gradecki 2003]
Gradecki, J. & Lesiecki, N. *Mastering AspectJ: Aspect-Oriented Programming in Java.* Indianapolis, IN: Wiley Publishing, 2003.

[Humphrey 1995]
Humphrey, W. *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.

[Kazman 2002]
Kazman, R.; O'Brien, L.; & Verhoef, C. *Architecture Reconstruction Guidelines, Third Edition* (CMU/SEI-2002-TR-034, ADA421612). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.
www.sei.cmu.edu/publications/documents/02.reports/02tr034.html.

[Kiczales 1997]
Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes, Cristina Videira; Loingtier, Jean-Marc; & Irwin, John. “Aspect-Oriented Programming,” 220-242. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP ’97).* Lecture Notes in Computer Science Volume 1241. ECOOP ’97, Jyvaskyla, Finland, June 1997. Springer, 1997.

[Kleppe 2003]
Kleppe, A.; Warmer, J.; & Bast, W. *MDA Explained, the Model-Driven Architecture: Practice and Promise.* Boston, MA: Addison-Wesley, 2003.

[Laddad 2003]
Laddad, R. *AspectJ in Action: Practical Aspect-Oriented Programming.* Greenwich, CT: Manning, 2003.

[Marinescu 2002]
Marinescu, F. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* New York, NY: John Wiley & Sons, 2002.

[Sun 2001]
Sun Microsystems. *Enterprise JavaBeans 2.0 Specification.* http://java.sun.com/products/ejb/2.0.html (2001).

[Xerox 2003]
Xerox Corporation. *The AspectJ Programming Guide.* www.eclipse.org/aspectj/doc/released/progguide/ (2003).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Using Aspect-Oriented Programming to Enforce Architecture	FA8721-05-C-0003

6. AUTHOR(S)

Paulo Merson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TN-019

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Using aspect-oriented programming (AOP), software developers can define customized compile-time error or warning messages that are issued when the code contains join points that match specified pointcuts. These customized messages are generated by compile-time declarations, which are an extremely simple but powerful AOP mechanism. Declarations that look for nonvalid interactions between modules can be used for architecture enforcement. Coding policies, best practices, design patterns, and code-naming conventions can also be enforced. Compile-time declarations operate as an additional verification in the build process, but they do not affect the compiled application and can be turned on and off at any time. That feature makes this approach an automated and nondisruptive solution for architecture enforcement and a risk-free first step towards AOP adoption.

14. SUBJECT TERMS	15. NUMBER OF PAGES
AOP, aspect-oriented programming, architecture enforcement, architecture conformance	24

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL